Research Institute for Advanced Computer Science
NASA Ames Research Center

BEYOND FORMALISM

Peter J. Denning

RIACS Technical Report 91.04

January 17, 1991

Beyond Formalism

Peter J. Denning

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report TR-91.4
17 Jan 91

The ongoing debate over the role of formalism and formal specifications in software features many speakers with diverse positions. Yet, in the end, they share the conviction that the requirements of a software system can be unambiguously specified, that acceptable software is a product demonstrably meeting the specifications, and that the design process can be carried out with little interaction between designers and users once the specification has been agreed to. This conviction is part of a larger paradigm prevalent in American management thinking, which holds that organizations are systems that can be precisely specified and optimized. This paradigm, which traces historically to the work of Frederick Taylor in the early 1900s, is no longer sufficient for organizations and software systems today. In the domain of software, a new paradigm, called user-centered design, overcomes the limitations of pure formalism. Pioneered in Scandinavia, user-centered design is spreading through Europe and is beginning to make its way into the United States.

This is a preprint of the column *The Science of Computing* for *American Scientist 79*, No. 1 (January-February 1991).

Work reported herein was supported in part by Cooperative Agreement NCC 2-387 between the National Aeronautics and Space Administration (NASA) and the Universities Space Research Association (USRA).

Beyond Formalism

Peter J. Denning

Research Institute for Advanced Computer Science

17 Jan 91

Among computer scientists there is a lively debate over the role of formalism in software engineering. One side argues that if formal methods were used more widely to prove that programs meet their specifications, we would suffer far fewer undependable programs and unsafe software systems. The other side argues that formal verification has not been proved practical, and that this situation is unlikely to improve soon. The two sides correspond roughly to the perspectives of mathematics and engineering.

The mathematical side is well represented by Edsger Dijkstra of the University of Texas at Austin, whose life work exemplifies elegant formal methods for designing correct programs. He has recently advocated that the introductory college course in computing avoid programming and concentrate on mathematics applicable to building correct algorithms *(1)*. David Gries of Cornell University argues similarly that complicated software structures could be routinely derived from their specifications by practitioners well versed in the notation and use of formal logic *(2)*.

The engineering side is well represented by Frederick P. Brooks, Jr., of the University of North Carolina at Chapel Hill, who is convinced that "the hard part of building software [is] the specification, design, and testing of the conceptual construct, not the labor of representing it and testing the fidelity of the representation." *(3)* He says that software is inherently complex because it does not have regular, repeated structures and because the state space of a software system--the set of all states it might occupy--is too large to comprehend directly. Brooks warns against putting too much faith in technologies that help simplify parts of the programming process without addressing the complexity question--technologies such as new high-level languages, object-oriented programming, artificial intelligence, expert systems, automatic programming, graphical program ming, program verification, environments and tools, and workstations. He says that major payoffs will result from buying software rather then building it anew, from rapid prototyping, from incremental development, and from cultivating Great Designers.

David Parnas of Queen's University says that the lack of competence with software is rooted in computer-science curricula, which do not prepare students well for real work in software engineering. Computing curricula, he says, have emphasized specialized topics that reflect the research interests of the founders of the discipline. He advocates a return to the basics of the traditional engineering curriculum *(4)*.

The Correctness Theorem

The technologies embodied in the functions of software systems rely on many formalisms: calculus, differential equations, discrete mathematics, linear algebra, probability, statistics, graph theory, numerical analysis, control theory, electrical circuit

theory, information theory, and signal processing. Even as these formalisms make software work, logic notation helps in the design of working software.

Nearly every computer scientist and engineer believes that a program or a software system must have a precise and unambiguous specification of exactly what it is supposed to do. Logic notation provides a language for such specifications. Many programmers would like some way to check whether the programs they produced meet the formal specification--that is, to prove the so-called correctness theorem: "Every result produced by this program is allowed by the specifications for the given input."

Performing such a check entails tracing the possible actions that a program can evoke; much of the debate over formal methods has focused on the feasibility of three basic strategies for this checking. One strategy is to use a theorem prover to trace through the program, enumerating logic formulas for all the intermediate lemmas and eventually determining whether or not the correctness theorem is true. Another strategy is to design programs within a development system that requires the programmer to specify a logic formula at the end of each code block, or other unit of program structure. Each formula must become true as a consequence of executing its associated block, and the formula at the end of the entire program must imply the specification. The third technique is to develop tests that only a correct program can pass.

Despite the diversity of views about the practicality of formal specifications, most commentators on the issue share the conviction that, in the end, the requirements of a software system can be unambiguously specified, and that meeting those specifications is an appropriate criterion for evaluating software. Most observers also agree that the design process can be carried out with little interaction between designers and users once

the specification has been agreed to. They share the vision of training software designers to regularly and systematically produce systems that are assessed by their users as helpful, relevant, reliable and dependable.

And yet hard questions persist. It is an old joke of software engineering that once users see the software in action, they exclaim, "Oh, it does what I said but not what I meant!" Why isn't this recognized in software-design practice? People adapt the way they work to the strengths and weaknesses of the software--so why do so many designers teach that the specification is fixed? Much of what people do is embedded in their routine daily practices and often is not obvious--so why do so many designers teach that the specification is formalizable at all?

Management

I see many similarities between approaches to software design and approaches to managing organizations. A love of formal specification permeates not only software engineering but also institutions and bureaucracies. A solution to the management problem might well make it possible for software engineers to use formalism more effectively, but more formalism is not going to solve the management problem. More formalism is not going to solve the software problem either.

By management I mean the discipline of forming, mobilizing, nurturing and guiding groups of people toward specific missions. It is a discipline of communication. Since nearly everyone works on a team or in an organization, good management is a matter of great concern. Our leading software engineers recognize this: Barry Boehm discusses how to design organizations capable of effective software production *(5)*; Gerald

Weinberg examines how to produce and nurture technical excellence in software teams *(6)*, and Brooks reminds us of Conway's Law, which says that systems tend to resemble the organizations that built them *(7)*.

Much practice for software design, however, ignores communication between designers and users. Software development is treated as a process of transforming formal specifications into programs that function correctly when executed. This model of software design appears frequently in textbooks and is the standard paradigm taught to students. It is a context-free model; it pretends that once the formal specification has been given, little or no communication needs to take place between those who will work with the system and those who design it.

In the first decade of this century Frederick Taylor introduced scientific management, a perspective founded on the assumption that work obeys scientific laws. He showed how factory operations could be decomposed into small tasks, with each task to be performed by one worker, and he claimed that for each task there was one best method that could be discovered by time-and-motion studies. The role of managers was to write up detailed formal descriptions of the tasks, to set the lowest price commensurate with the skill levels needed to carry out the tasks, and to find people whose personal characteristics matched them well to the jobs as described. Management's operational role was to supervise workers to be sure they performed tasks in the proper way and according to the master production plan created by managers.

This perspective led to major improvements in American business that produced American leadership in manufacturing lasting well into the 1960s. Over the past half-century, many Americans have come to accept this model as the one best method of

operating organizations and government. Managers routinely look for ways to formally specify our organizations with organizational charts, books of rules and procedures, formal job descriptions, and tests to match people optimally to available jobs. Communication from management is usually interpreted as instructions or orders. Formalism is also entrenched in the rules our government uses to specify how the bureaucracies work and how they should interact with private contractors--government gives formal specification, contractor delivers.

This observation is not my personal conclusion. Commenting recently on American business, Kososuke Matsushita, chairman of the Matsushita Electric Company in Japan, said: "We will win and you will lose. You cannot do anything about it because your failure is an internal disease. Your companies are based on Taylor's principles. Worse, your heads are Taylorized too. You firmly believe that sound management means executives on the one side and workers on the other, on the one side men who think and on the other side men who can only work. For you, management is the art of smoothly transferring the executive's idea to the worker's hands." *(8)*

Do you see the similarity between the paradigms of sofware and organizational design? And between communication between users and software designers, and communication between managers and workers? It is all very formal, context-free, and built on unidirectional communication. Japanese leaders see this as a major weakness, so ingrained that they can talk about it freely: We appear so Taylorized (formalized) as to be incapable of taking action to meet their challenge.

The essence of the limitation of formal, rule-based management is its inability to cope with rapid change, unexpected developments, and competition. The intelligence

and planning of managers and executives is insufficient for success in the global marketplace. Every employee must become a full participant in the mission of an organization and in creating innovations for the fulfillment of that mission. Tom Peters writes eloquently about management in a world of apparent chaos *(9)*.

The same limitations apply to software. This is obvious to those who see that, in practice, software is used to help organizations get their work done. Shifting expectations, rapid change, unexpected developments, and competition are as effective at confounding formal specifications for software as at confounding formalized management. The specifications often describe past conditions and prevent rapid adaptation to new conditions. Moreover, many important aspects of people's work are embedded in their everyday practice, where they escape the attention of designers writing specifications.

User-centered design

A new paradigm of software design has originated in Scandinavia under the leadership of Kristen Nygaard of the University of Oslo. It is gradually capturing the attention of software designers in Europe and the United States. It is called user-centered design and sometimes participatory design. It focuses on understanding the everyday practices of the people who will use the software, so that the software can be a useful, appropriate, dependable support for their work. In some ways this paradigm challenges the assumption enunciated by Brooks--that complexity is inherent in the software itself--by holding that the true source of complexity is not the internal structure of the software but the difficulty of understanding the essence of people's work.

Lucy Suchman reports on the work of Christiane Floyd of the University of Berlin, who, in a *festschrift* honoring Nygaard, outlined a paradigm change for software engineering *(10,11)*. Floyd contrasts a product perspective for software design with a process perspective, the former focusing on the derivation of a program from a specification and the latter focusing on the flows of work in an organization. Eleanor Wynn emphasizes that the way work is actually done can be found only in the daily practices, standards, and routines of the workplace, many of which are part of the shared understanding of the workers and are not explicitly stated as rules and principles *(12)*.

The process perspective holds that the commitments people make to one another are important; consequently software must help them track commitments to completion, and the software designer must achieve a deep understanding of the types of commitments that arise recurrently in a given organization *(13)*. The process perspective holds that software is not merely an artifact, but a part of the functioning organization; thus the work of defining objectives, establishing requirements, specifying functions, evaluating risks, making compromises, dealing with errors, and helping people learn to use and modify the system must be the work of an ongoing collaboration between software designers and users. It holds that the designer must understand the social context of the workplace and not attempt to abstract away from that context to purely information-processing aspects. It holds that people and machines have different roles that are often not interchangeable, and the job of the designer is to find and understand the difference. It holds that many of the errors arising in normal usage could be avoided if the designer understood the presuppositions and habitual expectations of the users.

You should not read the foregoing remarks as a call to drop formalism. On the contrary: formalism has demonstrated remarkable technological power. You should instead read that giving absolute priority to formalism limits what we can accomplish. We need to go beyond formalism and learn about communication in our organizations and in our software designs.

It is difficult for Americans brought up in the age of moon missions, interplanetary probes, supercomputing and biotechnology to go beyond love of principles and specifications. Nevertheless, the time has come to pay more attention to the murky, imprecise, unformalizable domains of everyday practice, which is, after all, where design is judged.

References

1. Edsger Dijkstra. 1989. On the cruelty of really teaching computing science. *Communications of the ACM 32*, 12 (December):1397-1414.

2. David Gries. 1991. Improving the curriculum through the teaching of calculation and discrimination. *Communications of the ACM 34*, 3 (March), to appear.

3. Frederick P. Brooks, Jr. 1987. No silver bullet: essence and accidents of software engineering. *IEEE Computer 20*, 4 (April):10-19.

4. David Parnas. 1990. Education for computing professionals. *IEEE Computer 23*, 1 (January):17-22.

5. Barry Boehm. 1981. *Software Engineering Economics*. Prentice-Hall.

6. Gerald Weinberg. 1986. *Becoming a Technical Leader.* Dorset House.

7. Frederick P. Brooks, Jr. 1975. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley.

8. Kososuke Matsushita. 1988. The secret is shared. *Manufacturing Engineering 100*, 2 (February).

9. Tom Peters. 1987. *Thriving on Chaos.* Harper & Row.

10. Lucy Suchman. 1988. Designing with the user. *ACM Transactions on Office Information Systems 6*, 2 (April):173-183.

11. Christiane Floyd. 1989. Out of Scandinavia: alternate approaches to software design and system development. *Human Computer Interaction 4*, 4.

12. Eleanor Wynn. 1991. Taking practice seriously. In *Design at Work: Cooperative Design of Computer Systems* (J. Greenbaum and M. Kyng, eds), Lawrence Erlbaum Associates.

13. Fernando Flores, Michael Graves, Brad Hartfield and Terry Winograd. 1988. Computer systems and the design of user interaction. *ACM Transactions on Office Information Systems 6*, 2 (April):153-172.